Dependable Distributed OSGi Environment

Miguel Matos
Universidade do Minho
mm@lsd.di.uminho.pt

António Sousa
Universidade do Minho
als@di.uminho.pt

ABSTRACT

As the concept of Service Oriented Computing matures the need for well defined architectures and protocols to address this trend is essential if IT is going to properly embrace SOC. The SOC paradigm has several requirements to work properly such as service composition and cooperation in a loosely coupled fashion, ability to adapt autonomously to environmental and business changes and address concerns such as modularity, dynamicity and proper integration between services. The popularization of the OSGi platform its another effort towards the SOC paradigm by issuing key aspects such as modularity and dynamicity in its service oriented design. However there is much room for improvement namely on the creation of architectures and mechanisms to improve the dependability of the overall solution by strengthening key properties such as the availability, reliability, integrity, safety and maintainability of the platform.

In this work we propose a middleware layer that offers the strong modular and dynamic properties required in an SOC environment by relying on OSGi while addressing dependability concerns. The starting point to achieve this is by instrumenting an OSGi implementation and providing means to monitor and manage it accordingly to business and environmental requirements. By relying on group communication facilities and some properties from the OSGi specification we are able to migrate OSGi environments between nodes thus minimizing service delivery disruption in the presence of faults and addressing, at the same time SLA properties by migrating (or shutting down) services that are consuming more resources than agreed/expected.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.2.11 [**Software Architectures**]: Domain-specific architectures

Keywords

SOC, OSGi, Virtualization, JAVA

1. INTRODUCTION

Service Oriented Computing is gaining momentum both in the academic and industrial worlds as can been seen from initiatives such as Amazon Web Services [2], and the popularity of OSGi platforms [13]. In SOC the customer buys a given service from the provider based on a Service Level Agreement that states the available resources and guarantees such as the persistence of the data stored, the dependability of the service and so on.

From the service provider point of view a major issue is strong isolation between customers and optimization of resources. The isolation has to be enforced at the data domain, that is the data of a given customer should only be accessed by itself, and also at the performance domain, the delivered service should not be constrained by the work being done by other customers. The way to achieve this has been through sand-boxing each customer in its own environment giving it the sense that he is the only one accessing the resources and/or services provided. This concept of virtualization is now strongly pervasive in most IT infrastructures as systems have to deal with many customers/users accessing the same physical resources. This solution offers proper data isolation as no customer/user is aware of others presence and also performance isolation by allocating proper physical resources to the virtual machines of each customer/user.

On the other hand the OSGi [13] platform is becoming very popular and counts several independent implementations such as Apache Felix [10], Eclipse Equinox [11], and Knopflerfish [12]. OSGi is a specification developed by major players on the IT field initially focused on solutions for embedded devices having a lightweight and well defined architecture. It consists of a service oriented architecture and its motto is "Dynamic Module System for the JAVA Platform".

It is modular because it is designed to split a system in partitions with well defined functionality and behavior that can be seen as a black box that provides a given set of functionalities through a well known API. This individual modules are called bundles in the OSGi terminology. By relying on already existing components and combining them, the effort of building a system is leveraged and due to its black box properties adding new functionality to an existing system could be achieved by adding a new bundle (or changing an existing one) without disrupting the production environment.

This leads us to the other key feature of OSGi: its dynamic properties. This characteristic allows bundles to be started, stopped, installed, uninstalled and updated in run-time with all the low level details of those operations addressed by the platform. This flexibility allows the architect to build a system in a plugin-like approach where parts of the system may be started and stopped on demand.

By relying on the service oriented principles of the OSGi platform as a starting point we intend to instrument it to have the notion of 'different' customers with different service requirements while guarantying isolation between them. To achieve this we need an Instance Manage capable of controling their life-cycle and enforce the different types of isolation needed. To address SLA concerns we need a Monitoring Module that is able to infer the state of the platform and of each one of the customers. With that knowledge and based on business policies the Autonomic Module shall be able to enforce the SLA by taking adequate actions such as stopping a bad behaved customer or migrating it to another node. This is indeed the major novelty of this work, the ability to migrate customers between physical nodes. The advantages of this approach are twofold: we are able to better respond to resource shortage on a given node by migrating the customer to a suitable node and we can cope with node failures by deploying the customer on an available node.

To summarize the goals of this work are the following:

1. Extend the OSGi Platform to be able to safely run multiple customers;
2. Ability to migrate customers between nodes;
3. Ability to measure resource usage of each customer;
4. Ability to enforce SLA requirements based on business policies.

The rest of the paper is organized as follows: in Section 2 we explain the instrumentation of the platform to support multiple customers; in the next Section 3 we present the different modules that address the remaining goals and finally we conclude on Section 4.

2. ARCHITECTURE

To accomplish the first goal, i.e, to be able to safely run multiple costumers on the same platform we need to provide isolation guarantees between each one of the customers and mechanisms to dynamically start and stop them to provision business needs. One way to accomplish this is to have one OSGi environment instance per customer and some kind of manager that is able to control all the OSGi instances available as we can observe in Figure 1.

Here we are just running multiple OSGi instances, each one on its own JVM that are controlled by some kind of external entity that we call the Instance Manager.With this approach we clearly achieve: namespace isolation between the customers as we are running them in separate JVMs; filesystem isolation by using a correctly configured SecurityManager, and probably performance isolation depending on the capabilities of the underlying Operating System and hardware. However this solution introduces much overhead because we are running multiple JVMs and it also difficults the task of managing the instances as we don't have a 'direct' method of accessing each one of them, we must rely on

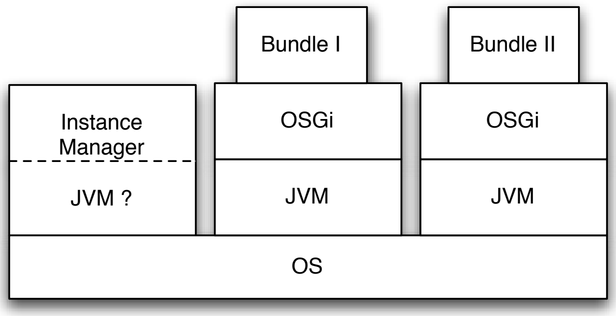

Figure 1: Running multiple OSGi instances on different JVMs.

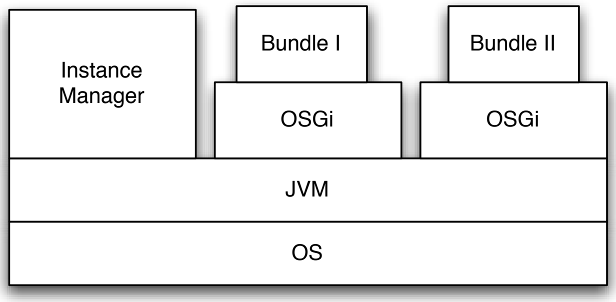

Figure 2: Running multiple OSGi instances on the same JVM.

communication methods like RMI, JMX, or TCP/IP connections further increasing the overhead and complexity of the solution.

It is possible to address most of these issues by bring all those instances under the same JVM as we can see in Figure 2. The overhead of multiple JVMs is gone and the management of the instances becomes simpler as we can easily start and stop embedded OSGi instances and maintain a simple data structure such as a Map to know about the existing instances and invoke operations on them.

However as we want this platform to be dynamic and modular and we are already using OSGi for those properties it makes sense to pull up the Instance Manager into the architecture stack and put it inside an OSGi environment. It is possible to observe this in Figure 3.

While this could be seen as a bloated design it is important to stress that it abides by the modularity and dynamic principles of OSGi and the Instance Manager could be seen as yet another bundle in the system. However the major advantage of this design lies in another interesting feature. By stacking OSGi instances on top of another OSGi environment we are able to make those instances 'communicate' with the underlying OSGi environment. In this setting, communicate means using services and packages provided by bundles running in the underlying OSGi environment. This design gives us the concept of virtual OSGI instances that are crafted to appear as normal OSGi environments to its client bundles but are able to use services provided by the underlying OSGi environment as it is possible to see in Figure 4.

If we compare Figures 3 and 4 it shall be possible to watch this concept in action. Supposing that the bundle named

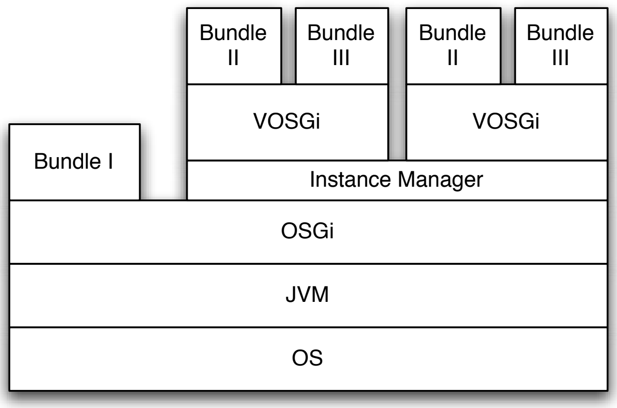

Figure 3: Running multiple OSGi instances inside an OSGi environment.

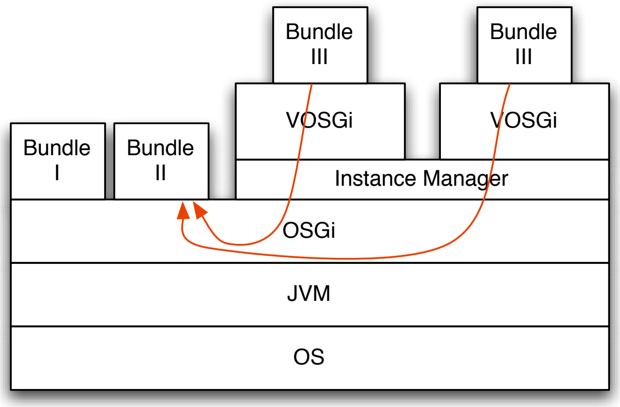

Figure 4: Multiple virtual OSGi instances using services from the underlying OSGi platform.

"Bundle III" depends on the bundle named "Bundle II" we could on the latter design pull down "Bundle II" to the underlying OSGi environment and make the instances of "Bundle III" running in the virtual instances use it in a transparent way. By using this technique it becomes possible to have only one instance of "Bundle II" whose services will be used by all the required bundles, either they are running on the same OSGi environment or in a virtualized one and, therefore leverage the management effort and optimize the resource usage of the platform.

Another important advantage is that this allows us to create virtual instances tailored to specific scenarios that nonetheless use services provided by the underlying OSGi framework. This is clearly a service oriented design as the customer (the user of the virtual instance) asks for a set of services and they are provisioned transparently by the underlying OSGi administrator freeing the customer to worry about details of that provisioning. General services, such as the log service, are well suited to be run in such a fashion.

On the technical side this is handled by passing references of the classloader of the underlying OSGi framework to the virtual instances that are in this way allowed to use the services and packages of the underlying framework. The potential leak of references among classloaders is prevented by the implementation of the framework (Apache Felix in this case) by carefully checking accesses to the exported classes.

In our implementation the services and packages to be exported to the virtual instances need to be explicitly indicated. This information is then used in a custom classloader that can be seen as the topmost classloader in the classloader's hierarchy of the virtual instance. When searching for a given class the virtual instance undergoes the normal lookup process and if this fails it checks the custom classloader. This classloader will then verify if the requested class has been explicitly exported, and if that holds true the request is passed to the underlying framework. By carefully instrumenting this feature and using an analogous approach to the services we ensure the safety of the solution i.e. no namespace and service references can be accessed without the explicit instruction of the administrator while providing a significant advantage in the management and resource usage of the whole platform.

To address isolation at the filesystem and network levels we rely on the SecuriryManager provided by the JAVA platform that should be configured by the administrator according to the business policies.

The key concepts of this architecture have been proposed before [7], to address isolation between different providers of digital services services to home that need to share the same box. Here we use barely the same architecture but apply those concepts in another environment and go further by offering mechanisms to improve the scalability and reliability of the solution.

3. CORE SERVICES

With the base architecture defined, we will describe in this section the modules that provide the services necessary to satisfy the other requirements such as the ability to enforce business policies, migrate and monitor OSGi instances.

Each service will be provided by a different module, an OSGi bundle, decoupled from the base architecture because it shall be possible to turn it on and off accordingly to the business needs. It is clear that some modules will need support from the above architecture to operate properly, namely with the Instance Manager, but it is a good design principle to separate this application logic from the core architecture keeping the platform as modular as possible.

In the rest of the section we describe each one of the modules and how they are instrumented to provide the required services.

3.1 Monitoring Module

The Monitoring Module shall be able to monitor the resource usage of each one of the running virtual instances and infer the overall resource availability, either in a dynamic way by inspecting directly the underlying operating system or by using some predefined values set up by the administrator. In this context the essential resources to monitor are memory usage, CPU time and possibly disk usage.

This is the least mature part of all the work developed as there are no adequate mechanisms to measure and monitor resource usage in the actual JVM specification [4]. This is particular true for the memory usage as the JVM only provides means to access the overall memory usage of the entire platform through `MemoryMXBean` objects. For the CPU usage it is possible to obtain a rough measure of that value on a per thread basis using the `ThreadMxBean` object. Combining this with `ThreadGroup` objects that allow to control several threads at the same time we are able to measure the CPU usage of a set of threads. In [15] the author combines that with Aspected Oriented principles to instrument the client bundles in a way that is possible to infer its CPU time consumption.

This solution is far from optimal as it requires an offline pre-processing of the bundle and leaves memory measurement outside the metrics. However, there is a proposed Java Specification Request 284: Resource Consumption API [4] that will address all the issues faced by providing a common framework to measure and manage resource usage in the JVM.

From the specification page: *Software systems in many circumstances need awareness of their resource usage. Meeting performance requirements often requires the ability to manage consumption of resources provided by the environment. Resource management is traditionally handled by operating systems, but the growing need to use the Java platform in the systems programming domain adds increased pressure to equip it with resource management capabilities at a level of abstraction that fits gracefully with the language.*

This JSR will clearly fill the gap that we pointed above by bringing awareness of resource usage to the JVM. The final draft has been in the workings for almost a year (August, 2007) and we are waiting for a reference implementation to start going on through this path.

3.2 Migration Module

This is the module that will be responsible to migrate the virtual instances from one node to another either instructed directly by the administrator or by the Autonomic Module.

This is the most challenging part of the overall design as several issues need to be addressed namely the following:

1. Knowledge of the available nodes and its resources;
2. Node failures.
3. State migration of the virtual instances.
4. Virtual instances services localization;

To address most of these issues in a dependable way we clearly need a group communication system (GCS) such as jGCS [3] as we have to have knowledge of the available nodes without relying on a centralized authority whatsoever.

Using a GCS and more particularly its membership service we have for free the knowledge of all the available nodes, and by exchanging messages with information about the virtual instances running on each node, we reliably address issue number 1.

When a notification about group membership changes is delivered this means that a node entered/leaved the group. In the former case there is nothing for this module to care about. Of course a global policy of balancing virtual instances among available nodes could be implemented but this shall be a concern of the Autonomic Module. In the case that a node leaves the system, this could be for two reasons: due to a 'normal' expected shutdown or due to a node failure. If it is due to a normal shutdown process, the Migration Module of that node migrates the virtual instances that need to continue providing the service to available nodes and the platform is shutdown. In the case of a node failure the Migration Module (of the remaining nodes) should use the knowledge about that node to redeploy the virtual instances among the available nodes in a decentralized way. The decision of where to redeploy the virtual instance shall take into account its resource requirements and the resources available on the destination node but these details will be handled by means of policies in the Autonomic Module. Instrumenting processes in this way we address concern number 2, about node failures and achieve graceful degradation in a overall setting. As we migrate instances of failed nodes to available nodes we continue to guarantee the delivery of the services provided by those instances despite a possible degradation of service due to resource constraints on the remaining nodes. The point to 'how much to degrade', for example by refusing to accept more virtual instances past a given threshold or just swap or stop virtual instances to accommodate one with higher priority shall also be defined through business policies and therefore addressed by the Autonomic Module.

To address issue 3 is important to stress some points. We assume a underlying SAN[1] or distributed filesystem to ensure that data written by each node is accessible globally. This is mainly to focus on the problem we are trying to solve as replication of data at this low level is a research topic for itself. The other important point is that the OSGi [13] specification enforces that the framework state shall be persistent across framework reboots. Here state means the information associated with the life-cycle of the bundles in the framework, namely which ones are installed and its running state (started, stopped, etc).

Combining the above we are able to transparently and quickly redeploy any virtual instance because the state of the framework is made persistent per the OSGi specification and available network-wide by the assumption made about the underlying storage mechanisms. The cost of this operation is therefore comparable to a normal startup of the platform, probably less, as we already have the basic services deployed on the underlying framework.

However we still need to address the state transfer of the bundles running inside the virtual instances because the specification only address the state persistence of the framework, not the state of the bundles running inside it.

Traditionally services (bundles in this case) could be either stateless of stateful. In the former case there is no state associated with the bundle and therefore (re)starting it on the target instance is enough to ensure the continuation of service delivery. In the later case we have the persistent state accessible by the other nodes, as we assumed a SAN mechanism, and the same procedure of (re)starting the bundle on the target node could be applied. Additionally to the persistent state stored on disk, we also need to address the running state (context) of the bundle such as stack frames, state of objects and threads and so on. In stateless services, as each request contains all the necessary information to be processed, it is common practice to resend the request until it is addressed and therefore the running context of the bundle is not as important as in a stateful bundle. In that case the running context is important to properly handle subsequent (causal related) requests and it is closely related to application semantics. If the application provides transactional mechanisms then the client could be informed about the outcome of the request (a typical example being database systems) and therefore this case could be reduced

[1]Storage Area Network

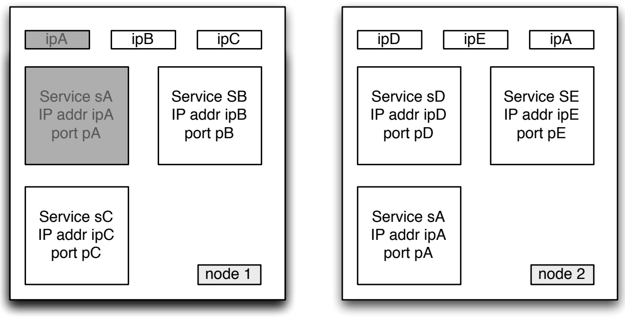

Figure 5: Services with unique IP address and Communication port.

to the stateless example as the request contains all the information to be processed without leaving the system in an inconsistent state should the request fail. If those properties are not provided then the application should be able to cope with and resolve inconsistencies, as there is no external generic mechanism able to ensure this, as it is closely related to the application semantics. In the future we intend to address this by further instrumenting the platform to be able to lively migrate the running context of the bundles as it was been proposed in several approaches [14, 1, 8, 9].

With that ability we do believe that we can approach near zero downtime of services by, for example, having the running context of the bundle replicated on other nodes and doing instantaneous failover in case of node failures. Naturally this approach has many issues to solve, namely the costs and feasibility of strategies such as the pointed above but the approach seems worth investigating in this scenario.

Each one of the virtual instances, and possibly some of the services running in them, need to be accessed through the network to be useful. An Internet available service is characterized by its IP address and communication port. Different services running at a single node cannot share the same IP address and communication port, although one of them might be shared.

Having an IP address per service although optimal may be impractical, at least in IPv4, due to the lack of available IP addresses. In such a scenario migrating a service from a node to another one simply requires the node currently holding the service to release the IP address, and the new node to bind it to one of its network interfaces as depicted in Figure 5.

The most common scenario would be one where several services share a single IP address, having each one of them a unique communication port. In this scenario, depicted in Figure 6, migrating the IP address along with the service may not be an option as the IP address may be in use by other services. In such a scenario, it might be useful to decouple the IP address from the service and use an external service such as a fault tolerant IP virtual server (ipvs). The ipvs will be responsible to ensure the availability of the IP address to the Internet and redirect the service requests to the node currently running the service. Notice that this setting allows also to scale-up the services allowing multiple instances of the service and use the ipvs as a load balancer.

Using one of the above strategies we address issue 4, of service localization. It is also important to note that the IP and Communication port could be attributed to a service

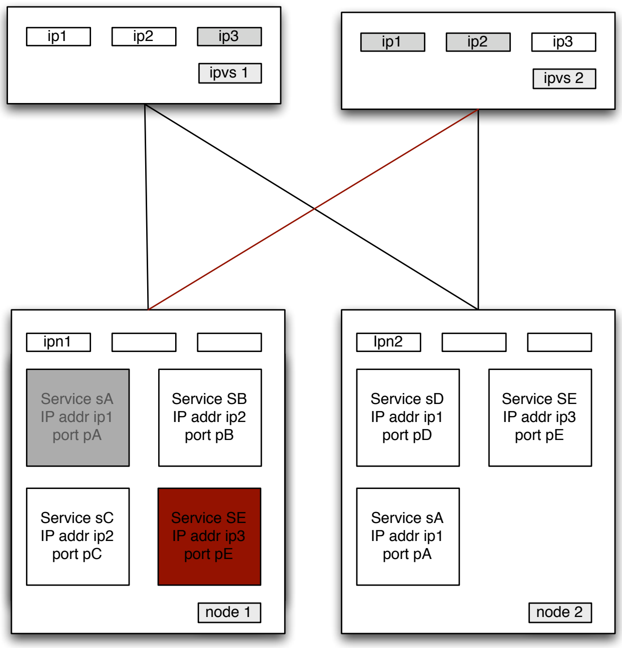

Figure 6: Services with shared IP address and unique Communication port.

provided by a bundle running inside a virtual instance or to the instance itself depending on business needs. The former case is useful for example to provide access to a web service, where the later could be useful if the customer needs full access to an OSGi platform. Additionally if the IP and Communication port is associated with a virtual instance we also must ensure that bundles running on that instance could only bind to that IP address.

3.3 Autonomic Module

The Autonomic Module shall enforce the business policies defined by the administrator. This may include stopping a given virtual instance, giving it lower priority if it is consuming more resources than agreed and swap it, if possible, to a suitable node capable of properly address the instance requirements while optimizing global resource usage. It is also the responsibility of this module to take the proper measures in the presence of faults to prevent service delivery disruption.

By using the Monitoring Module to build the view of the system and the Migration Module to know about other nodes (and eventual failures and restarts) the Autonomic Module is able to enforce the business policies. For example it is able to instrument the Migration Module to migrate a given instance due to resource constraints or handle events provided by the same module to address a node failure by (re)deploying an instance that was running on the failed node. In a local setting this module could be used to fine tune a given instance or bundle by adjusting its parameters accordingly to environmental and business changes.

This module relies on an existing framework, Serpentine [6], which is an adaptive middleware for heterogeneous distributed systems that is ready to run as an OSGi bundle. Although with a simple architecture it provides interesting features such as being stateless, having hierarchization capabilities and allowing the policies to be defined in a programmatic approach by means of the Scripting for the Java Platform [5]. The cascading capabilities allow instances of the module to be composed on each other and therefore supporting different levels of control of the system by hiding unnecessary or unwanted details on different hierarchies.

4. CONCLUSIONS

In this paper we proposed a software architecture capable of safely run multiple costumers in an OSGi environment while addressing some dependability concerns. The safety of the solution is achieved by sand-boxing each one of the customers in its own virtual instance and therefore addressing the different isolation concerns between them.

Nonetheless the proposed architecture aimed to be flexible to allow efficient and rational usage of resources. This is achieved by allowing service composition between the virtual instances and the underlying OSGi environment, thus allowing it to have a single instance of base services that can be used by each one of the virtual instances. We have also achieved this goal using the migration module, which allows to concentrate in a single node, several customers when they are idle or requiring limited resources, and relocating them in another node when they need more performance. Using this approach we not only ensure better usage of the resources available at each node, but also reduce power usage by shutting down or hibernating nodes when they are not needed. This can be regarded as a side effect of our implementation that, nonetheless, contributes to the reduction of the green house effect.

Fault tolerance and scalability are also issues that must be addressed in any SOC architecture. In our case, the proposed architecture and services allow the quick recovery of failed nodes, simply by restarting the services running in the failed node in another(s) node(s). By using an ipvs infrastructure we are also able to cope easily with service migration and service scalability. We may start as many replicas of the service as required and the ipvs infrastructure can, to some extent, transparently perform load-balancing thus scaling the service performance beyond the performance of a single node.

Finally we were also able to enforce service level agreements based on business policies. This allows the characterization of the situations where more resources are needed or reduced, thus allowing for optimal resources usage. Allowing each business to define its own SLA requirements is an important feature as it improves the flexibility of the architecture not limiting its applicability to a single domain.

From an implementation point of view the work on the base architecture is almost completed and we already tested it by running multiple virtual instances that use services from the underlying environment namely the log service, the HTTP service and the JMX server service. The same holds true for the Autonomic Module due to being an already existing OSGi-enabled component. Work on the Monitoring Module is stalled for technical limitations and we are maturing the Migration Module to start testing the whole platform. We also need to stabilize the communication interfaces of each one of the components but this is happening naturally as all the components fall in their place.

5. REFERENCES

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: a language for resource-aware mobile programs. In *Mobile Object Systems Towards the Programmable Internet*, pages 111–130. Springer-Verlag, 1997.

[2] Amazon. Amazon web services. http://aws.amazon.com/, 1996-2008.

[3] N. Carvalho, J. Pereira, and L. Rodrigues. Towards a generic group communication service. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 1485–1502. Springer Berlin / Heidelberg, 2006.

[4] G. Czajkowski. Resource consumption management api. Technical report, Java Community Process (JSR'284), 2007.

[5] M. Grogan. Scripting for the java platform. Technical report, Java Community Process (JSR'223), 2006.

[6] M. Matos, A. Correia, J. Pereira, and R. Oliveira. Serpentine: adaptive middleware for complex heterogeneous distributed systems. In *ACM symposium on Applied computing*, pages 2219–2223. ACM, 2008.

[7] Y. Royon, S. Frénot, and F. L. Mouel. Virtualization of service gateways in multi-provider environments. In *Component-Based Software Engineering*, pages 385–392. SpringerLink, 2006.

[8] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in java. In *Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pages 16–28. Springer-Verlag, 2000.

[9] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *ACM symposium on Operating Systems Principles*, pages 68–77. ACM, 1995.

[10] The Apache Software Foundation. Apache felix. http://felix.apache.org/.

[11] The Eclipse Foundation. Equinox. http://www.eclipse.org/equinox/.

[12] The Knopflerfish Project. Knopflerfish. http://www.knopflerfish.org/.

[13] The OSGi Alliance. Osgi service platform. http://osgi.org/osgi_technology/download_specs.asp, Aug. 2005. Release 4.

[14] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in java. In *Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agent*, pages 29–43. Springer-Verlag, 2000.

[15] I. Yamasaki. Monitoring and managing resource usage on osgi frameworks. In *OSGi World Congress*, 2005.